Efficient GitHub Crawling using the GraphQL API*

Adrian Jobst, Daniel Atzberger, Tim Cech, Willy Scheibel[0000−0002−7885−9857], Matthias Trapp[0000−0003−3861−5759], and Jürgen Döllner

Hasso Plattner Institute, Digital Engineering Faculty,
University of Potsdam, Germany
{adrian.jobst, daniel.atzberger, tim.cech, willy.scheibel,
matthias.trapp, juergen.doellner}@hpi.uni-potsdam.de

Abstract. The number of publicly accessible software repositories on online platforms is growing rapidly. With more than 128 million public repositories (as of March 2020), GitHub is the world's largest platform for hosting and managing software projects. Where it used to be necessary to merge various data sources, it is now possible to access a wealth of data using the GitHub API alone. However, collecting and analyzing this data is not an easy endeavor. In this paper, we present Prometheus, a system for crawling and storing software repositories from GitHub. Compared to existing frameworks, Prometheus follows an event-driven microservice architecture. By separating functionality on the service level, there is no need to understand implementation details or use existing frameworks to extend or customize the system, only data. Prometheus consists of two components, one for fetching GitHub data and one for data storage which serves as a basis for future functionality. Unlike most existing crawling approaches, the Prometheus fetching service uses the GitHub GraphQL API. As a result, Prometheus can significantly outperform alternatives in terms of throughput in some scenarios.

Keywords: Mining Software Repositories · *GitHub* Crawling · GraphQL API · Microservices · Event-driven.

1 Introduction

Today's software development projects are characterized by a large number of stakeholders and a comprehensive technology stack. The activities of the stakeholders and their communication with each other are systematically stored in various software repositories, be they code repositories, source code control

* We want to thank the anonymous reviewers for their valuable feedback to improve this article. This work is part of the "Software-DNA" project, which is funded by the European Regional Development Fund (ERDF or EFRE in German) and the State of Brandenburg (ILB). This work is part of the KMU project "KnowhowAnalyzer" (Förderkennzeichen 01IS20088B), which is funded by the German Ministry for Education and Research (Bundesministerium für Bildung und Forschung).

repositories, bug repositories, archived communications, or artifacts from builds, testing, or deployment. The Software Analytics domain generates interesting and actionable insights about the software development process by analyzing the extensive data available in software repositories [34]. For example, machine learning methods are used that require training data. This data is usually extracted from online software repositories as they contain large amounts of openly available data. Various systems have been developed to generate datasets from publicly available software repositories. There are already large collections of raw or enriched software data that can be used for studies [14,28,6]. In addition, end-to-end approaches have also been proposed to support scientists and engineers throughout the process, from data collection to analysis [7,31]. However, although existing datasets contain large amounts of data, they may be incomplete, the pre-processing steps may be unclear or obstructive, or the chosen format may be inappropriate.

In this work, we propose Prometheus, a system that is based on an event-driven microservice architecture. In a microservice architecture, functionality is separated into services. This separation reduces the need to understand implementation details of existing functionality, as new services communicate via REST-like API's, events, or a mixture of both. Being able to connect to event streams also facilitates the development of live analytic scenarios, which is especially interesting for industry use cases [26]. Prometheus consists of two components, a GitHub crawling service and a storage service that is important to overcome API throughput limitations. We chose the GitHub project management platform as it contains a large number of software artifacts, i.e., source code control information, software development information, and metadata like developer information and comments. The latter two are less represented in Mining Software Repositories (MSR) research [8]. Through the redesigned GitHub GraphQL API, Prometheus can access all information available on GitHub and retrieve it in less time than other systems that use the prior REST API.

The remainder of this work is structured as follows: In section 2 we review existing work related to our approach. We provide an overview of our concept in section section 3. In Section 4 we present a detailed description of our system and provide implementation details. The experimental setup and the results of our evaluation are presented and discussed in section section 5. We conclude this paper in section 6 and present directions for future work.

2 Related Work

Various systems have been proposed which acquire data from online repositories. Linstead et al. presented Sourcerer, an infrastructure for analyzing source code repositories [22]. The authors processed source code files from GitHub projects and analyzed them using Latent Dirichlet Allocation [2] and its variant, the Author-Topic Model [29]. The results may serve as a summary for program function, developer activities and more. A system developed to support scientists and practitioners in MSR research is Boa, presented by Dyer et al. [7]. The

system provides the infrastructure and a domain-specific language for accessing large software repositories such as GitHub. Researchers can access the service and create datasets for their experiments through a web-based interface. Another system for generating large datasets of open source projects was presented by Ma et al. [23]. The authors created World of Code, a very large and frequently updated collection of version control data for open source software projects that is updated monthly. In their work, the authors explore several issues related to the structure of open source projects. Trautsch et al. presented the SmartSHARK ecosystem to support MSR research [32]. It consists of four parts, which are tools for crawling data from different hosting platforms, e.g., GitHub or Jira, a storage system, a web application that can be used for data collection and a web application that provides an overview of the collected data. While the aforementioned systems combine crawling and processing functions, other systems focus more on crawling to create datasets that can be used as a research base. Gousios and Spinellis presented the GHTorrent project [14]. Ultimately, GHTorrent aims to create an offline mirror of GitHub. To this end, it retrieves event data from the GitHub REST API, from which the original GitHub data schema has been reconstructed, and makes everything available in database dumps. In a follow-up study, Gousios presented a dataset created by GHTorrent [15], and in another follow-up study, Gousios et al. described a new feature to obtain customizable data dumps on demand [16]. Another mentionable project in this context is GH Archive (www.gharchive.org/), which is an open source project to record and archive the public GitHub timeline (event data). GH Archive data is often used as a data basis in MSR related research [17,1,12].

In general, a deeper understanding of data sources and potential use cases helps in the design and development of MSR systems. Kalliamvakou et al. analyzed the data quality of GitHub data in more detail [20]. The study showed that there are some dangers to be aware of when using GitHub data for research or analysis purposes. For example, 40 % of all pull requests do not appear as merged, even though they were. Also, many projects are personal or inactive. This could be a potential threat for analysis purposes Munaiah et al. addressed the issue and developed a classifier to identify whether a repository is a developed software project or not [27]. Another study that analyzed GitHub metadata was conducted by Borges et al. In their work, they investigated what factors influence the popularity of GitHub repositories [3].

As previously described, systems often use REST API's of repository hoster to gather data. But in the case of GitHub, there is also a GraphQL based alternative. Due to the growing popularity and adoption of GraphQL in general, academia has also turned to the topic. Motivated by the lack of a formal definition, Hartig and Pérez formalized the semantics of GraphQL queries and then analyzed them [18]. This allowed them to prove certain things, such as that the GraphQL evaluation problem is NL-complete and that GraphQL answers can be prohibitively large for Internet scenarios. Wittern et al. studied 16 commercial GraphQL schemas and 8,399 GraphQL schemas to investigate GraphQL interfaces. They found that the majority of APIs are vulnerable to denial of ser-

vice through complex queries, which also introduces security risks [33]. A direct comparison between the REST and GraphQL architectural models was made by Seabra et al. [30]. Three target applications were implemented in both models, from which performance metrics could be derived. Two-thirds of the applications tested saw performance improvements in terms of average number of requests per second and data transfer rate when using GraphQL. But in general, performance was also below that of its REST counterpart when the workload exceeded 3000 requests. Similarly, Brito et al. migrated seven REST-based systems to use GraphQL [4]. The migration reduced the size of JSON responses by up to 94% in the number of fields and 99% in the number of bytes (median results). In another study, Brit and Valente described a controlled experiment in which students had to implement similar queries, once in REST, once in GraphQL [5]. Their results showed that students were faster at implementing GraphQL queries, especially when the REST endpoints contained more complex queries. Surprisingly, the results held true for the more experienced groups of graduate students as well. As mentioned earlier, GraphQL queries can be unexpectedly large, not least because of their nested structure, making query cost estimation an important feature. Estimating costs based on a static worst-case analysis of queries has had limited success, leading Mavroudeas et al. to propose a machine learning-based approach. Testing their approach on publicly available commercial APIs, they found that their framework is able to predict query costs with high accuracy and consistently outperforms static analysis.

The presented systems come with limitations when used for data generation. Some systems have obstructive data transformations of raw data [22] or provide a fixed set of processing functionality [7] which excludes them as usable tools when a different transformation or format is needed. Systems which are able to provide raw data [14,32] do usually only leverage a subset of available data, e.g. issue tracking and version control history. Additionally most systems use the GitHub REST API, whose rate limit is rather conservative which introduces the need for multiple API tokens, otherwise crawling additional information takes long. By using the GraphQL API, a single API token can crawl significantly more information, which makes it more suitable for most users.

3 Concept

The Prometheus architecture follows an event-driven microservice architectural style. At first, the implemented system will consist of two components. A fetching component to crawl data from the GitHub GraphQL API, and a database component that stores the responses in a database management system. As there is no common formal definition of microservices and event-driven applications, we will start by outlining important aspects of these, before describing the fetching component in section 3.2 and database component in section 3.3.

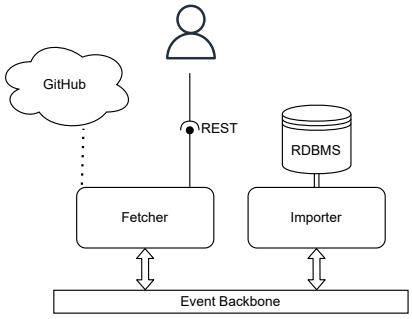


Fig. 1: General concept of the Prometheus system architecture.

3.1 Architecture Considerations

According to Fowler and Lewis, a microservice architectural style means that an application is developed as a suite of small services. These services usually communicate via lightweight mechanisms and are built around business capabilities. Furthermore, each service runs in its's own process, which means that they can be deployed independently [10]. Figure 1 shows the general concept of the Prometheus System, including two services, one for crawling and one for persisting crawled data. Fowler and Lewis also described the main characteristics of microservices derived from practice, since there is no formal definition. Some of them also apply to our system.

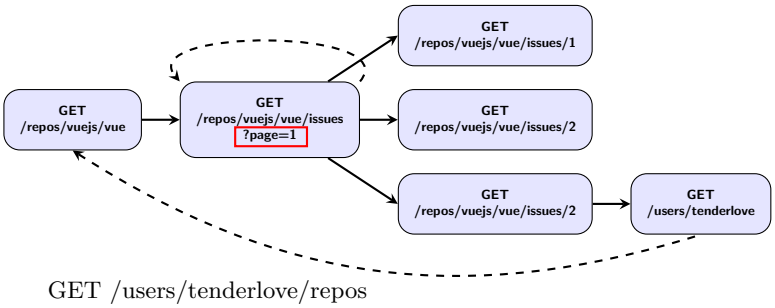
In software engineering, a component usually refers to a unit of software, that encapsulates a certain functionality, e.g., via a library. In monolithic applications, components are tightly coupled to the code that uses them, thus making a complete redeployment necessary when a component is changed. The microservice approach mitigates this problem by dividing the application into services, each of which can be deployed individually. So in the architecture seen in Figure 1, a change related to the importing service would not affect the fetching service at all. In monolithic applications, components communicate via method invocation or function call which execute in-process. Microservices use a coarser-grained approach where the communication medium should be as simple as possible. Usually, an HTTP request-response or lightweight message bus is used for asynchronous communication with routing, which is also used in our system to control the crawling component. The right tool for the right job is a philosophy that microservice architectures follow. Unlike centralized governance, where standardization of certain technologies leads to restrictions on the choice of development tools, developers of microservice architectures choose the tools they need to create a particular service. When one wants to extend the Prometheus system, the only technology restriction will be the component that manages services. While a single data model seems reasonable, it is often not realistic, as different components may have slightly different views of the data. It may be useful for each service to use its own appropriate conceptual model, which has the advantage that each service can use a database system that is most suitable, also known as

"polyglot persistence" [21]. In our system, the Importer service stores the raw data collected from GitHub, and new services, even if they build on this data, may use a new storage component. We believe this also has usability benefits, as single conceptual models are easier to understand than a complicated mixture.

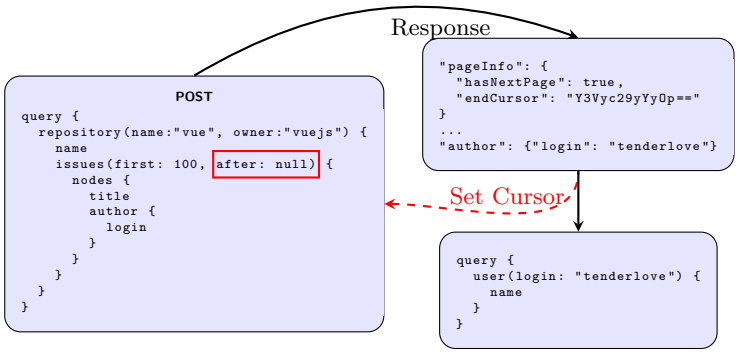
Event-driven architectures are also not formally defined, most likely because there are many different notions of what an event is and what it is used for. An event contains at least the form or action of the event and a timestamp of when the event occurred. In summary, three purposes of events can be formulated [11]. In the simplest case, an event just serves as a mere notification that something happened. For instance, in our system, the fetching service publishes data from the crawling process as events. As mentioned earlier, two systems or services, even if they use conceptually different data models, can still have similar attributes. In this case, state changes from one service must be transmitted to others that have a similar view of the data which can be done in the form of an event. In our system, the importing service publishes state changes via events so that dependent or related components can react to them. For a small monolithic application with a single data model, the state of the application is usually mirrored in the database or can be derived from it in case of failure. In contrast, the state of a distributed system is mirrored in all data of all services together. However, because services can change state individually or event-driven state transfers can occur, accessing older state is not as easy as with a single database log. One way to maintain this capability in a distributed system is event sourcing, where all changes to the system are recorded as events [9]. In theory, the event log can be used to recover the system state at any point in time. This feature will not be used in the current version of Prometheus but may be relevant for future use-cases where it can be implemented without any architectural change.

3.2 GitHub Fetcher

The crawling component uses the GraphQL API from GitHub to fetch metadata. Mainly because the rate limit is more liberal than in the case of the REST API. But also because one GraphQL call can replace multiple REST calls and therefore returns more data. But using the GraphQL paradigm also imposes differences in crawling logic which can be seen in Figure 2. For instance, when crawling metadata for a repository and its issues on GitHub, the entry point is the according `/repos/` URL path as seen in Figure 2a. The response of such a call then contains URLs to linked entities, e.g. repository issues. Paginated endpoints usually just include summary information of entities, so in order to get all information, subsequent calls for every single entity must be made. With GraphQL, all these entities can be included in a single call which is shown in Figure 2b. Since we need a cursor to fetch more than 100 results, the crawling is more complex making it difficult to get a parallel connection. Furthermore, responses do not contain information about linked entities, so in the case of Figure 2b, the system must infer how to formulate a query that fetches user information of issue authors.



(a) REST API crawling. Pagination information is encoded in URL parameters.



(b) GraphQL API crawling. A cursor from the response is needed to paginate.

Fig. 2: A comparison of how entities of the GitHub REST and GraphQL API's are crawled. Red parts specify pagination settings.

Since it is good practice to combine services that change frequently and the GitHub data schema may change often [10], our service could also address data persistence. We argue that it is unlikely that there will be frequent changes to the GitHub data model as GitHub has a long history of how people use the API and has put a lot of thought into the GraphQL schema. So it is more likely that the data model just gets extended for new features, which will not break the existing ones. GraphQL also makes it easy to handle minor changes, for example, attributes that will soon be dropped can be marked as deprecated, and new changes can simply be added as nullable attributes.

Therefore, storage capabilities will be handled explicitly by a separated service as described in section 3.3.

The interface of this service will only be used to control the crawling process which means sending crawling jobs and monitoring progress. In our system job descriptions will be defined as GraphQL queries. Using GraphQL for job descriptions makes sense since it is the same paradigm as the GitHub API, and because it seems easier for developers to form GraphQL queries instead of e.g. REST queries [5]. These queries will be the same as regular GitHub GraphQL API

queries but allow for additional or different parameters. For instance, the query seen in figure 2b uses 100 as the first argument value, which is the allowed maximum in the GitHub API. In our system values beyond the allowed value, e.g. 10.000 are possible. By allowing additional or changed arguments, the system needs logic on how to handle them. For example, it needs to be able to paginate automatically when more than 100 elements are requested. An additional problem is when we nest connections and require more than 100 elements in the nested connection. Pagination requires a cursor of the previous page, which is specified as the after argument as seen in figure 2b. The response to a query with nested pagination returns multiple cursors for the nested entities since the cursor refers to the parent entities. But since there can only be supplied one cursor, there is no correct way to map these in a consecutive query. But in most cases, the nested pagination problem can be solved by resolving it in separate queries. This can be achieved by replacing a connection field with a field that queries just one entity. In the left query of figure 2b, *issues* can be replaced with *issue* which returns only a single issue. This would resolve nesting if any, but requires splitting the query into one that fetches all issues and a second one that paginates the nested connection.

3.3 Database Importer

The sole purpose of the database import service is to capture the data published by the GitHub fetching service and store it in a database. An important design decision is the choice of database, as it has a large impact on the schema, database interface, and transactional properties.

The GraphQL API provides a schema for GitHub's data model so it is normalized to some extent since the schema provides the relationships between structured entities. In some cases, it seems tempting to store this data in a denormalized form in a document store. For instance, if repository entities are stored, the corresponding topics can simply be stored in a nested list in a document store. Bulk loading this data could be faster since no join operations are required. However, this would result in the repetition of data, and the increased storage requirements that would result may not be feasible since GitHub provides huge amounts of data. Since, if a topic name changes, every occurrence in all objects must be changed, it is more challenging to keep the data consistent. Moreover, it is not possible to nest every type of entity in this way, as the size of the documents would become very large. Also, some relationships would still need to be modeled, such as the relationship between repositories and their forks. A mixture of relationships and nested properties would lead to a cluttered data model and complicated application logic and a performance decrease [25].

Finally, no caching strategy is implemented. This means that even if entities were already fetched and stored in the database, they will always be fetched again in new queries. Systems that rely on the REST API often use such a strategy, in the GraphQL context it makes less sense. This is because in the REST API ETags can be used. ETags are GitHub's REST APIs way to determine if a resource has changed. This makes it easy to check in advance if an entity stored

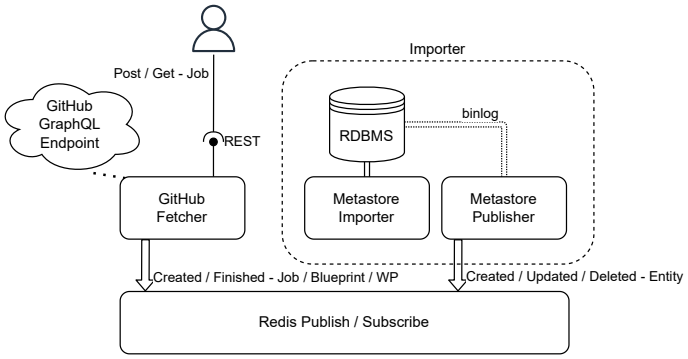


Fig. 3: Prometheus system architecture docker setup. Event subscriptions are excluded for clarity.

in the database needs an update. In GraphQL, one could check the `updatedAt` field, but unlike the REST API, this consumes rate limit, so one could also simply ask for all attributes and update the entity if needed.

4 Prometheus System

The individual services are developed and deployed as Docker containers. These coincide with the conceptual requirements for microservice architectures. In fact, it has been shown that Docker can be a good fit when implementing microservices [19]. Figure 3 shows the general Docker setup of Prometheus. The importer service consists of two containers, and one for importing fetched entities, one for publishing changes in the database. Separating service functionality in several containers is not uncommon. This way the system can spawn multiple containers of the desired functionality in case of a heavier workload. Both use the same relational database. The fetching functionality resides in one single container. The GitHub fetcher is using the GitHub GraphQL endpoint to query data. Querying jobs can be submitted via a REST API, as well as getting job progress summaries. The GitHub fetcher as well as the metastore publisher are publishing events via an event service when service state changes occur. Redis is chosen as the event system, whose basic publish/subscribe as well as a queuing mechanism is sufficient for a prototype implementation of the proposed architecture.

GitHub Fetcher. One of the most important functions of the crawling service is the processing of the job definitions. This includes splitting the query if there is nested pagination. In addition, pagination must continue until the parameters entered by the user are satisfied or there are no more objects. It must also pass parameters from responses to consecutive queries resulting from the splitting process if any. If a query has nested pagination, the way to resolve it is to first query the top paginated node, e.g. issues of a repository, and then for every node in the response query the nested nodes, e.g. assignees of every single issue.

Listing 1.1: Nested pagination split algorithm

```
queries = [initial_query]
consecutive_info = []

current_query = initial_query

while True:
  actual_query, info = remove_nested_pagination(current_query)

  if actual_query != current_query:
    remove_field_nodes(queries[-1], info.obsolete_nodes_left)
    remove_field_nodes(actual_query, info.obsolete_nodes_right)

    queries.append(actual_query)
    consecutive_info.append(info.consecutive_info)
  else:
    break

  current_query = actual_query
```

This approach is not always applicable, more precisely it is only possible if the returned entities of the paginated top node can also be accessed directly.

Pseudocode on how to do that can be seen in code listing 1.1. The algorithm starts with the query originally supplied, then `remove_nested_pagination` replaces connection nodes containing a nested connection with their direct-access counterpart (e.g. issue instead of issues). It is important that the top node is replaced and the rest of the branch remains untouched, even if it contains more nested connections. If a substitution has taken place, the nested pagination will be removed from the previous query (e.g. assignees of issues). Also, obsolete nodes are removed from the new query, i.e. all connection nodes that do not have a nested connection. These are already crawled by the previous query and are not needed in the new query. Finally, the new query and follow-up parameter information, e.g. an issue number, are added. The loop continues to replace nested connections until there are none left.

5 Evaluation

This section evaluates the performance of *Prometheus* fetching and import services. As a performance metric, we will measure throughput in fetched and stored entities per second. This is done for Prometheus and *Microsoft's ghcrawler*, a REST-based GitHub crawler, to see if the promised speed increase through the GraphQL API is true. Two experiments are performed, one simple and another with deeper relationships.

5.1 Simple fetching scenario

In the simple fetching scenario, the *vuejs/vue* repository (github.com/vuejs/vue) and all the issues it contains are crawled. The repository contains 327 open issues and 9 370 closed issues. So, theoretically, 9 698 entities are being fetched

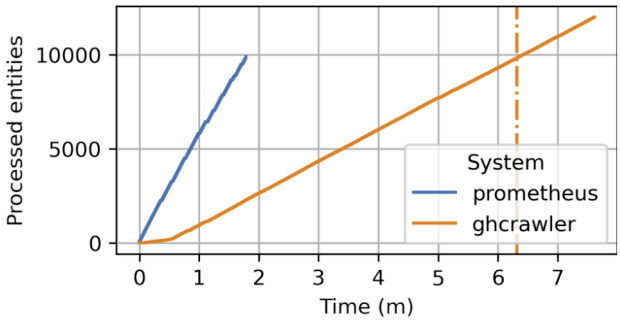


Fig. 4: Comparison of crawling performance between Prometheus and ghcrawler. In the GitHub REST API, every pull request is an issue, so the issues endpoint return issues and pull requests. The vertical line indicates at what point pure issues (without pull requests) would have been completed. Considering that, Prometheus is still 3.5 times faster.

(9 697 issues plus 1 repo). Currently, there is only one adjustable parameter for retrieval performance in Prometheus, which is the number of work packages that are combined in a query. Combining more work packages increases performance, but can also lead to timeouts. Currently, this parameter cannot be dynamically adjusted and is set to 100 for both experiments. This is quite aggressive and sometimes leads to API timeouts, but so far never to unresolvable timeouts. For ghcrawler, the required visitor maps are implemented in the source code. Four tokens and ten concurrent processing loops are used for both tasks.

Figure 4 shows the result of the simple job. The first thing to notice is that ghcrawler processes more entities than Prometheus. This is because the GitHub REST API considers each pull request as an issue, but not vice versa [13]. Thus, the REST API also returns all pull requests in the issues endpoint, and therefore ghcrawler processes them (2 169 additional entities). The GitHub GraphQL API explicitly separates pull requests and issues, so there is no such overhead in Prometheus job execution. A unique feature of this job in Prometheus is that the work packages are completely sequential. Getting the next page of Issues requires the last cursor, which means there is no advantage to combining work packages. In terms of overall job execution, Prometheus is 4.2 times faster, with an average throughput of 92 entities per second, while ghcrawler has a throughput of 26 entities per second. If we assume that ghcrawler does not retrieve the unwanted pull requests, Prometheus is still 3.5 times faster at retrieving all entities. On average, the processing loop of the Prometheus fetching service took 1.14 seconds with a standard deviation (SD) of 0.46 to process a work package. The majority of that time is used to make the actual API request, which took 1.09 seconds (SD = 0.46) on average.

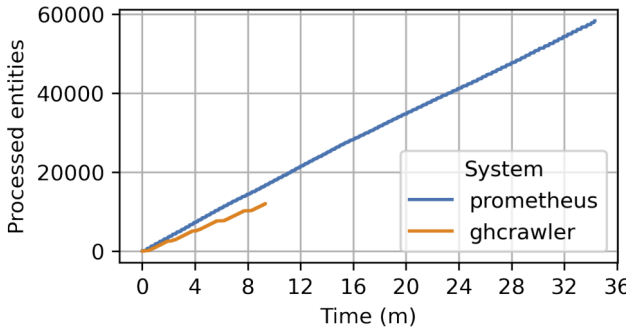


Fig. 5: Comparison of crawling performance between Prometheus and ghcrawler on a job with nested pagination. Prometheus takes 3.7 times longer because a large number of consecutive work packages are generated, which increases fetching and processing time. However, more entities are still fetched per unit time.

5.2 Complex fetching scenario

In the complex fetching scenario, the repository with all issues is also fetched, but in addition, the assigned users of the issues are also captured. In this case, the number of additional entities cannot be estimated in advance.

Figure 5 shows the result of the complex scenario. This time Prometheus took significantly longer than in the previous example, 3.7 times longer than ghcrawler. The reason is that the current implementation of Prometheus strictly separates nested queries. So there is one query that retrieves a page with 100 issues, which results in 100 new individual queries for each issue that paginates the connection of the assignees. So in this example, Prometheus has 9 794 queries (97 + 9 697) to fetch. In the actual processing loop, up to 100 queries can now be combined per call. Therefore, only two more calls to the API are required for this job than for the previous job. However, since the queries are now more complex, one processing loop now takes 19.48 seconds (SD = 2.69) on average. The actual API calls take 6.12 seconds (SD = 0.48). Although most of the combined queries have about twice the number of nodes and the same number of points as the queries in the previous example, the API calls take about 5.6 times longer. The loop execution time is also longer due to the need to decompose the response, extract more cursors, and create more consecutive work packages.

Prometheus also has significant overhead in this situation. We still only crawl a single repository, but it is fetched in each of the 9 794 queries. In addition, we now fetch 222 entities related to accounts, 97 repository owners when fetching issue pages as in the last example, and 125 users assigned to issues. Of these 222 accounts, only 20 are unique. Since only 125 issues have a user assigned, 9 572 queries return only an empty connection to assignees, making them redundant. Overall, Prometheus throughput in this scenario relates to 28 entities per second. Ghcrawler, on the other hand, behaves almost exactly like the previous example with a throughput of 21 entities per second. This is because the issues endpoint

response already returns the summary representation of the assignees. Also, there are only 20 unique users, so previously queried users that exist in the database are not queried again because they are retrieved from storage.

5.3 Discussion

In the simple use case – fetching all issues from a repository – Prometheus clearly outperforms ghcrawler in both execution time and token consumption. Even if we exclude the discussed pull request overhead when retrieving issues, Prometheus is still 3.5 times faster when retrieving all issues. Looking at token consumption, the difference is drastic. While ghcrawler requires at least three tokens to fetch all requests from the REST API, Prometheus consumes only about two percent of the rate limit of one token. The result of the second experiment is different, Prometheus is slower in this case. This is because the current implementation incurs unnecessary overhead when fetching empty connections. Also, at the moment it is fully synchronous in terms of the actual API calls. Although Prometheus is 3.7 times slower, it still has higher throughput and lower token consumption. The token consumption has remained almost the same even though the queried nodes were much more, which is since queries could be combined in this query. This is a particularly interesting result, as another study suggests that GraphQL API responses are smaller than those of REST API's [4]. While this is true for an end-user application, crawling application developers must be careful to avoid this pitfall. Especially because GraphQL may perform worse on heavier loads than a REST counterpart [30], one does not want to flood GraphQL endpoints with unnecessary calls. Even if not present in this experiment, the opposite can also be the case, badly chosen calls can lead to unexpectedly complex queries which may either overload the server or even the client [24].

6 Conclusion

In this work, we presented Prometheus, a system for crawling software repositories from GitHub at scale. We demonstrated that an event-driven microservice architecture is applicable in the context of mining software repositories. Traditional systems used the REST API, but the new GraphQL API promises a significant throughput and token consumption advantage. To show whether these promises hold, we compared Prometheus, which uses the GraphQL API, to Microsoft's ghcrawler, which is based on the REST API. The throughput achieved by Prometheus is higher in all test scenarios, while token consumption is significantly lower. The job execution time is 3.5 times faster in a simple scenario but 3.7 times slower in a more complex scenario due to the currently synchronously implemented fetching loop.

Future Work. Originating from the performed experiments, we plan to make future upgrades to Prometheus. One is using more elaborate queries. Instead of

the strict splitting, we can still query items from the nested connection. When querying paginated fields, one can also retrieve the total count of items available. If we do both, we can eliminate all the redundant queries that made the second experiment slow. We can also further optimize for throughput. For example, a consecutive query always fetches parents of the paginated node of interest, resulting in severe overhead that should be omitted. Furthermore, the processing loop should be asynchronous so that the high response times do not affect the execution time. Lastly, more use cases have to be implemented and tested to verify the system's effectiveness.

References

1. Bjertnes, L., Tørring, J.O., Elster, A.C.: LS-CAT: A large-scale CUDA AutoTuning dataset. In: 2021 International Conference on Applied Artificial Intelligence (ICAPAI). pp. 1–6. IEEE (2021). https://doi.org/10.1109/ICAPAI49758.2021.9462050
2. Blei, D.M., Ng, A.Y., Jordan, M.I.: Latent dirichlet allocation. Journal of Machine Learning Research **3**, 993–1022 (2003)
3. Borges, H., Hora, A., Valente, M.T.: Understanding the factors that impact the popularity of github repositories. In: 2016 IEEE International Conference on Software Maintenance and Evolution (ICSME). pp. 334–344 (2016). https://doi.org/10.1109/ICSME.2016.31
4. Brito, G., Mombach, T., Valente, M.T.: Migrating to GraphQL: A practical assessment. In: Proc. 26th International Conference on Software Analysis, Evolution and Reengineering. pp. 140–150. SANER '19, IEEE (2019). https://doi.org/10.1109/SANER.2019.8667986
5. Brito, G., Valente, M.T.: REST vs GraphQL: A controlled experiment. In: Proc. International Conference on Software Architecture. pp. 81–91. ICSA '20, IEEE (2020). https://doi.org/10.1109/ICSA47634.2020.00016
6. di Cosmo, R., Zacchiroli, S.: Software heritage: Why and how to preserve software source code. In: iPRES 2017-14th International Conference on Digital Preservation. pp. 1–10 (2017)
7. Dyer, R., Nguyen, H.A., Rajan, H., Nguyen, T.N.: Boa: Ultra-large-scale software repository and source-code mining. ACM Transactions on Software Engineering and Methodology **25**(1), 1–34 (2015). https://doi.org/10.1145/2803171
8. de F. Farias, M.A., Novais, R., Júnior, M.C., da Silva Carvalho, L.P., Mendonça, M., Spínola, R.O.: A systematic mapping study on mining software repositories. In: Proceedings of the 31st Annual ACM Symposium on Applied Computing. pp. 1472–1479. SAC '16, ACM (2016). https://doi.org/10.1145/2851613.2851786
9. Fowler, M.: Event sourcing (2005), `https://martinfowler.com/eaaDev/EventSourcing.html`, accessed on 17.05.2022
10. Fowler, M., Lewis, J.: Microservices (2014), `https://www.martinfowler.com/articles/microservices.html`, accessed on 17.05.2022
11. Fowler, M.: What do you mean by "Event-Driven"? (2017), `https://martinfowler.com/articles/201701-event-driven.html`, accessed on 17.05.2022
12. Gasparini, M., Clarisó, R., Brambilla, M., Cabot, J.: Participation inequality and the 90-9-1 principle in open source. In: Proceedings of the 16th International Symposium on Open Collaboration. pp. 1–7. ACM (2020). https://doi.org/10.1145/3412569.3412582

13. Github: List repository issues, `https://docs.github.com/en/rest/reference/issues#list-repository-issues`, accessed on 17.05.2022
14. Gousios, G., Spinellis, D.: GHTorrent: Github's data from a firehose. In: Proc. 9th International Workshop on Mining Software Repositories. pp. 12–21. MSR '12, IEEE/ACM (2012). https://doi.org/10.1109/MSR.2012.6224294
15. Gousios, G.: The GHTorrent dataset and tool suite. In: Proceedings of the 10th Working Conference on Mining Software Repositories. pp. 233–236. MSR '13, IEEE (2013). https://doi.org/10.1109/MSR.2013.6624034
16. Gousios, G., Vasilescu, B., Serebrenik, A., Zaidman, A.: Lean GHTorrent: GitHub data on demand. In: Proceedings of the 11th Working Conference on Mining Software Repositories. pp. 384–387. MSR 2014, ACM (2014). https://doi.org/10.1145/2597073.2597126
17. Hagiwara, M., Mita, M.: Github typo corpus: A large-scale multilingual dataset of misspellings and grammatical errors. arXiv preprint arXiv:1911.12893 (2019)
18. Hartig, O., Pérez, J.: Semantics and complexity of GraphQL. In: Proc. World Wide Web Conference. pp. 1155–1164. WWW '18, International World Wide Web Conferences Steering Committee (2018). https://doi.org/10.1145/3178876.3186014
19. Jaramillo, D., Nguyen, D.V., Smart, R.: Leveraging microservices architecture by using docker technology. In: SoutheastCon 2016. pp. 1–5 (2016). https://doi.org/10.1109/SECON.2016.7506647
20. Kalliamvakou, E., Gousios, G., Blincoe, K., Singer, L., German, D.M., Damian, D.: The promises and perils of mining github. In: Proceedings of the 11th Working Conference on Mining Software Repositories. pp. 92–101. MSR '14, ACM (2014). https://doi.org/10.1145/2597073.2597074
21. Leberknight, S.: Polyglot persistence (2008), `http://www.sleberknight.com/blog/sleberkn/entry/polyglot_persistence`, accessed on 17.05.2022
22. Linstead, E., Bajracharya, S., Ngo, T., Rigor, P., Lopes, C., Baldi, P.: Sourcerer: mining and searching internet-scale software repositories. Data Mining and Knowledge Discovery **18**(2), 300–336 (2009). https://doi.org/10.1007/s10618-008-0118-x
23. Ma, Y., Bogart, C., Amreen, S., Zaretzki, R., Mockus, A.: World of code: An infrastructure for mining the universe of open source vcs data. In: Proc. 16th International Workshop on Mining Software Repositories. pp. 143–154. MSR '19, IEEE/ACM (2019). https://doi.org/10.1109/MSR.2019.00031
24. Mavroudeas, G., Baudart, G., Cha, A., Hirzel, M., Laredo, J.A., Magdon-Ismail, M., Mandel, L., Wittern, E.: Learning GraphQL query cost. In: Proc. 36th International Conference on Automated Software Engineering. pp. 1146–1150. ASE '21, IEEE/ACM (2021). https://doi.org/10.1109/ASE51524.2021.9678513
25. Mei, S.: Why you should never use mongodb (2013), `http://www.sarahmei.com/blog/2013/11/11/why-you-should-never-use-mongodb`, accessed on 17.05.2022
26. Menzies, T., Zimmermann, T.: Software analytics: So what? Software, IEEE **30**, 31–37 (2013). https://doi.org/10.1109/MS.2013.86
27. Munaiah, N., Kroh, S., Cabrey, C., Nagappan, M.: Curating github for engineered software projects. Empirical Software Engineering **22**(6), 3219–3253 (2017). https://doi.org/10.1007/s10664-017-9512-6
28. Ortu, M., Destefanis, G., Adams, B., Murgia, A., Marchesi, M., Tonelli, R.: The JIRA repository dataset: Understanding social aspects of software development. In: Proceedings of the 11th International Conference on Predictive Models and Data Analytics in Software Engineering. pp. 1:1–4. PROMISE '15, ACM (2015). https://doi.org/10.1145/2810146.2810147

29. Rosen-Zvi, M., Griffiths, T., Steyvers, M., Smyth, P.: The author-topic model for authors and documents. In: Proc. 20th Conference on Uncertainty in Artificial Intelligence. pp. 487–494. UAI '04, AUAI Press (2004)
30. Seabra, M., Nazário, M.F., Pinto, G.: REST or GraphQL? a performance comparative study. In: Proc. XIII Brazilian Symposium on Software Components, Architectures, and Reuse. pp. 123–132. SBCARS '19, ACM (2019). https://doi.org/10.1145/3357141.3357149
31. Tiwari, N.M., Upadhyaya, G., Rajan, H.: Candoia: A platform and ecosystem for mining software repositories tools. In: 2016 IEEE/ACM 38th International Conference on Software Engineering Companion (ICSE-C). pp. 759–761 (2016)
32. Trautsch, A., Trautsch, F., Herbold, S., Ledel, B., Grabowski, J.: The SmartSHARK ecosystem for software repository mining. In: Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering: Companion Proceedings. pp. 25–28. ACM (2020). https://doi.org/10.1145/3377812.3382139
33. Wittern, E., Cha, A., Davis, J.C., Baudart, G., Mandel, L.: An empirical study of GraphQL schemas. In: Proc. International Conference on Service-Oriented Computing. pp. 3–19. Springer (2019). https://doi.org/10.1007/978-3-030-33702-5_1
34. Zhang, D., Han, S., Dang, Y., Lou, J.G., Zhang, H., Xie, T.: Software analytics in practice. Software, IEEE **30**, 30–37 (09 2013). https://doi.org/10.1109/MS.2013.94